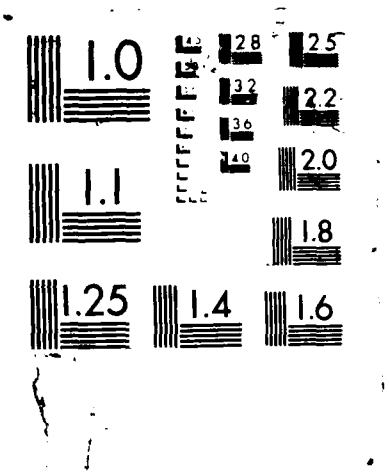
AD-A190 885 VPC - A PROPOSAL FOR A VECTOR PARALLEL C PROGRAMMING LANGUAGE(U) ILLINOIS UNIV AT URBANA CENTER FOR SUPERCOMPUTING RESEARCH AN.. V A GUARNA 30 OCT 87 1/1

UNCLASSIFIED CSRD-666 AFOSR-TR-87-1969 F49620-86-C-0136 F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

Unclassified

SE

(2)

# ORT DOCUMENTATION PAGE

1a. AD-A190 885

1b. RESTRICTIVE MARKINGS DTIC FILE COPY

2a. ...AUTHORITY

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution unlimited.

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CSRD Report No. 666

5. MONITORING ORGANIZATION REPORT NUMBER(S)
AFOSR-TR- 87-1969

6a. NAME OF PERFORMING ORGANIZATION
The Board of Trustees of the University of Illinois

6b. OFFICE SYMBOL (If applicable)

7a. NAME OF MONITORING ORGANIZATION
AFOSR/NM

6c. ADDRESS (City, State, and ZIP Code)
506 S. Wright St.
Urbana, IL 61801

7b. ADDRESS (City, State, and ZIP Code)
AFOSR/NM
Bldg 410
Bolling AFB DC 20332-6448

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
AFOSR

8b. OFFICE SYMBOL (If applicable)
NM

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
~~AFOSR~~ F49620-86-C-0136

8c. ADDRESS (City, State, and ZIP Code)
AFOSR/NM
Bldg 410
Bolling AFB DC 20332-6448

10. SOURCE OF FUNDING NUMBERS

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO. |
|---|---|---|---|
| 61102F | 2304 | A3 | |

11. TITLE (Include Security Classification)
VPC - A Proposal for a Vector Parallel C Programming Language

12. PERSONAL AUTHOR(S)
Vincent A. Guarna, Jr.

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| Submitted Paper | FROM 10/1/86 TO 9/30/87 | Oct 30, 1987 | |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | |
| | | | |
| | | | |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper proposes a definition for VPC, an extended C programming language for vector-parallel applications. VPC is a superset of the conventional C language that contains extensions for vector and parallel machines. New constructs and their semantics are presented, along with some discussion about potential problems that arise when extending C into the parallel domain. The reader is assumed to be familiar with the C programming language--this paper only describes those aspects of VPC that differ from the standard definition.

(Keywords: parallel processing; synchronization).



20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
D

22a. NAME OF RESPONSIBLE INDIVIDUAL
Maj. John P. Thomas

22b. TELEPHONE (Include Area Code)

22c. OFFICE SYMBOL
NM

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

CSRD report no. 666

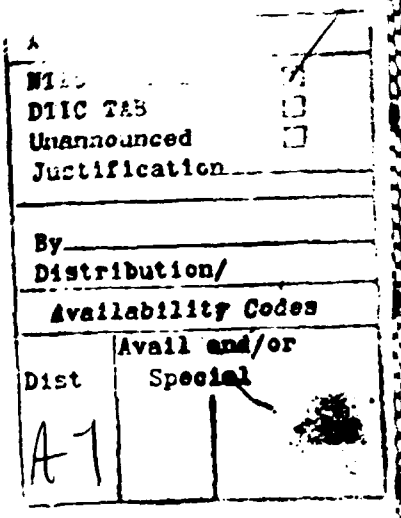


*Center for*
*Supercomputing Research and Development*


# VPC – A Proposal for a Vector Parallel C Programming Language

Vincent A. Guarna, Jr.

June 16, 1987




University of Illinois at Urbana–Champaign
104 S. Wright Street
Urbana, Illinois 61801

CSRD report no. 666

# VPC – A Proposal for a Vector Parallel C Programming Language

Vincent A. Guarna, Jr.

June 16, 1987

*Abstract*

This paper proposes a definition for VPC, an extended C programming language for vector–parallel applications. VPC is a superset of the conventional C language that contains extensions for vector and parallel machines. New constructs and their semantics are presented, along with some discussion about potential problems that arise when extending C into the parallel domain. The reader is assumed to be familar with the C programming language — this paper only describes those aspects of VPC that differ from the standard definition.

Keywords: C language, vector processing, parallel processing, synchronization, parallel programming languages

This work was supported in part by the National Science Foundation under Grants No. US NSF DCR84-06916 and US NSF DCR84-10110, the US Department of Energy under Grant No. US DOE DE-FG02-85ER25001, the US Air Force Office for Scientific Research under Grant No. AFOSR-85-0211, and by a donation from the IBM Corporation.

# CONTENTS

# 1. OVERVIEW

## 1.1. Purpose

This paper presents a proposal for the definition of Vector Parallel C (VPC), a C programming language for vector multiprocessors. With many parallel machines appearing in the marketplace, individual vendors are devising special methods for exploiting parallelism on their products. Since many software application development environments typically support more than one vendor's equipment, there is a strong incentive to attempt to define a standard language environment in order to promote portability. Although VPC is not likely to become the standard for parallel C environments, some of the ideas and problems presented should be of value to those who will be commissioned to develop an official specification.

VPC is designed to be an extended version of the C language as defined by Kernighan and Ritchie (Ref. 8). Rather than taking the approach of extending programming language functionality through the use of system calls, VPC extends the syntax of C to support the explicit expression of vector and parallel constructs. Although parallel programming environments for C have been built using library routines (Refs. 2, 14), serious users frequently bypass these facilities and resort to assembly language programming in order to eliminate excess overhead. An efficient compiler combined with a sufficiently expressive language should obviate that necessity.

## 1.2. Existing Extended C Environments

Much work has been done with respect to extending the C programming language. One effort is the definition of Vector C by Kuo-Cheng Li at Purdue University (Refs. 11, 12). Originally implemented on a Control Data Cyber 205, Vector C supports language constructs for vector processing. Although Vec-

tor C defines no constructs for parallel execution, it represents a thorough extension of the C programming language which supports an orthogonal set of vector constructs for existing C arithmetic and logical operators. Many Vector C ideas have been directly incorporated into the design of VPC.

Another parallel C programming environment is C*, developed by the Thinking Machines Corporation (Ref. 17). Designed for the Connection Machine (Ref. 7), C* is an extension of C that supports parallelism through the use of parallel objects. By introducing minimal syntactic extensions, C* supports a mechanism for parallel execution on vector-style data. C* uses new storage classes to declare parallel objects.[1] Operations that are performed on parallel objects (by conventional C constructs and operators) are automatically executed in parallel. A selection mechanism allows the programmer to control the extent of the parallelism.

EPEX/C is another extended C language, developed at the IBM T.J. Watson Research Center, Yorktown Heights (Ref. 5). EPEX/C is implemented with a preprocessor that accepts a parallel C syntax as its input language and translates it into standard C syntax as its output. EPEX/C defines no vector constructs, but supports a flexible structure for concurrency control. Originally targeted at the RP3 project (Ref. 15), EPEX/C includes type declarations for private and shared data, as well as constructs for parallel execution of loop and non-loop code sequences. EPEX/C also defines an extensive set of library calls for message passing and interprocess synchronization.

Some parallel C environments support parallelism control through a library of system calls. Sequent and Alliant both support environments for their shared memory multiprocessors that are built on system calls (Refs. 14, 2). These calls include the conventional Unix[2] fork() function for initiating parallelism (Ref. 9). They also provide functions for sharing memory between processors, as well as functions for supporting interprocess synchronization through the use of indivisible memory operations. Sequent supports parallel execution of iterative loops on multiple processors through a microtasking facility on the Balance series of multiprocessors (Refs. 14, 16). Alliant also provides parallel loop execution support through func-

[1] Parallel objects (defined as "poly" in C*) are allocated on a one-per-processor basis.

[2] Unix is a trademark of AT&T Bell Laboratories.

tion calls that activate the FX/8's proprietary concurrency hardware (Refs. 2, 3).

### 1.3. Philosophy

The following sections describe a set of extensions to C that provides the ability to access the functionality of a multiple processor system. The general philosophy of the C language is to generate concise code and provide flexibility while overlooking potential errors pertaining to type inconsistency and statement structure. VPC remains consistent with that ideology with its language extensions. VPC allows things that are plausible, ignoring data dependences whenever possible (Refs. 4, 10, 19). Additionally, just as standard C environments provide lint (Ref. 18) as a separate utility to perform more rigorous semantic checking of serial programs, VPC environments should provide a similar tool for parallel programs to check semantics with respect to data–dependence analysis. VPC only intervenes in those cases where a clear–cut error has been made (such as the passing of a private variable as a parameter to another task).

Where possible, the syntax and semantics for VPC have been designed with a machine–independent attitude — there are no constructs that specifically require a particular machine organization.[3] VPC compilers should ascribe some specific run–time behavior to various constructs in a deterministic way, allowing vendors to provide access to proprietary architectural features of their machines while remaining compatible with a standardized language model. Such machine–specific implementations should be supplied with sufficient user documentation to allow interested users to exploit the architectural aspects of a particular system.

[3] However, many of the constructs discussed are efficiently implemented on shared memory multiprocessors. VPC was originally designed in the context of this machine organization.

# 2. DECLARATIONS

## 2.1. Extensions

VPC extends the traditional type declarations with a new modifier called an **access class specifier**. Access class specifiers are used to control the sharability of data objects declared for use in VPC. The following new elements are added to the set of C reserved words to accommodate access class specifiers:

**private**
**shared**
**sync**

Syntactically, the access class specifier is an optional keyword that, when present, must precede the storage class and type specifiers for the data declaration. Two combinations of access class and storage class specifiers are illegal. These are **shared register** and **sync register**. These combinations are flagged as an error by the VPC compiler.

Examples:

```
int x;                              /* defaults to shared automatic  */
private int x;                      /* defaults to automatic         */
shared float y[100];                /* defaults to auto              */
shared automatic float y[100];      /* identical to previous decl.   */
shared static float y[100];
sync float a[10][10];
```

## 2.2. The PRIVATE Storage Class

Identifiers declared with the **private** storage class are defined to be visible only to the processor which allocates them. Though the **private** declaration does not affect any of the normal C scoping rules for single task applications, it does affect visibility in multiple task applications. Identifiers declared as **private** are generally allocated on the local processor stack as is customary with conventional C compilers. More details on **private** identifiers and scoping idiosyncrasies are given in sections 4 and 5.

## 2.3. The SHARED Storage Class

Identifiers declared with the **shared** storage class are defined to be sharable by all processors running on a particular application. **Shared** identifiers are allocated in a system area that is accessible by all processors (either directly or indirectly). Although **shared** identifiers are located in a globally accessible area of memory, standard C scoping rules could conceal their visibility from some processors. Detailed examples are given later.

Care must be taken when using shared pointers in VPC. Specifically, pointers declared to be **shared** may point to data declared as **shared** or **private**. However, loading shared pointers with the addresses of private data could cause erroneous results. For example:

```
shared  int     *x;
shared  int     y;
private int     z;

x = &y;
x = &z;
```

The first assignment loads a globally accessible pointer **x** with the address of a globally accessible integer, **y**, and functions identically for all tasks. The second assignment loads a globally accessible pointer with the address of a private identifier. Each task that dereferences **y** accesses the same location in its virtual address space. However, accesses by all tasks other than the one that allocated **x** are unpredictable. VPC generates a compile-time warning for the second assignment.

## 2.4. The SYNC Storage Class

Identifiers declared as **sync** are meant to be used for interprocessor synchronization and communication. For this reason, **sync** variables are generally associated with a set of indivisible operations. VPC supports a set of atomic primitives that preserve integrity during update operations, but **sync** variables are additionally protected by the compiler for normal C assignment statements and unary operators. Assignments to and unary operations on **sync** variables are guaranteed not to conflict with any atomic synchronization primitives supported by the system. Examples and more details are given in section 4.4.

## 2.5. Defaults

With one exception, all data declarations in VPC that do not explicitly specify an access class are assigned the **shared** storage class. While this might tend to increase the probability of anomalous program behavior through inadvertent side effects, it is more conducive to the development of communication–intensive parallel application programs. Multitasking programs, by default, are permitted to share data.[4] This should allow maximum compatibility with existing C semantics and require a minimal amount of special coding by the programmer to provide access to shared variables. The exception is **register** variables which default to the **private** access class.

Note that the data sharing attribute is completely independent of the scope for a given identifier. A datum that is sharable is not necessarily global in scope. Consider the following example:

```
main()
{
        int     x;
          .
          .
        spawn a(x)
          .
          .
}

a(y)
        int     y;
{
        int     x;
          .
          .
        spawn a new process with x
          .
          .
}
```

In this example, routines **main()** and **a()** both have an identifier named **x** that is sharable. However, there is no conflict in the global name space. Main's **x** is a separate allocation (and therefore physical memory

[4] For shared memory multiprocessors, this usually means that data is allocated in global memory. Since shared global memories tend to require longer access times than local memories, VPC compilers would be expected to allocate only potentially sharable portions of activation records in global memory to maximize run–time performance. For non–shared memory machines, failure to do this optimization would be disastrous.

location) from a's declaration. With code generated by conventional C compilers, these two identifiers would have distinct positions in the activation records of their associated routines. The only difference in the parallel domain is that the activation records are located in global memory (thus allowing the potential for sharing).

The **shared** default storage class policy means that VPC programs using extensive parallelism have the potential to create many inadvertent side effects through shared variables. To accommodate those users who prefer compiler–enforced protection against this possibility, two new directives to the C preprocessor are added. One is **#private**, which specifies that all type declaration statements that lexically follow it are to be given the **private** storage class by default. The other is the **#shared** directive, which returns the compiler to its standard default of giving the **shared** attribute to unspecified type declarations.

# 3. VECTOR CONSTRUCTS

## 3.1. Vector Declarations

Vectors in VPC are declared in the usual C style for array objects:

```
float x[100];
double y[10][10];
```

Anything declared as an array may be operated on with any legal vector operations.

## 3.2. Vector References

Vector operations are explicitly requested by the programmer through the use of a specific vector reference syntax. This vector syntax is similar to the Fortran 8X (Ref. 13) syntax – it consists of lists of subscripts of the form *starting_element* : *ending_element* : *stride*. The specification of the stride is optional and, if missing, is assumed to be one (the second colon must also be omitted for this default case). Either or both of *starting_element* and *ending_element* may be missing. If *starting_element* is missing, it is assumed to be the beginning of the array (always 0 in C). If *ending_element* is missing, it is assumed to be the last element in the array. If both are missing (in which case the use of the colon is optional), the entire array is assumed. Note that *ending_element* may not be omitted for arrays which are dynamically allocated nor for formal parameters.

Here are some examples:

```
float a[100], b[200], c[300];

/* Example 1 */     a[0:99] = b[0:199:2];
/* Example 2 */     a[:] = b[::2];
/* Example 3 */     a[] = b[];
/* Example 4 */     a = b;          /* Illegal – see below */
/* Example 5 */     a[1:10] = b[1:20:2] * c[1:30:3];
/* Example 6 */     a[] = c[] * b[];
```

Example 1 shows a simple vector assignment. The reference to the a vector completely specifies all of the

elements. The reference to the b vector completely specifies alternating elements. Example 2 shows a semantically equivalent assignment, but with incompletely specified subscripts. Example 3 shows the most abbreviated syntax for vector references. This statement, however, indicates a non-conformable vector assignment. In keeping with the policy of VPC, executable code will be generated by the compiler and the operation will proceed as requested. In such cases where the left- and right-hand sides are not conformable, the shape of the left-hand side dominates the assignment. Example 3 copies the first 100 elements of the b vector into a and then terminates. This results in a vector instruction that is equivalent to the following FOR loop:

```
for (i = 0; i < 100; i++)
    a[i] = b[i];
```

Example 4 is illegal. Because C allows the programmer to specify the base address of an array (or vector) by indicating the array name only (or array name with less than the defined number of subscripts), using the array name alone to specify an entire vector[5] creates ambiguity in the semantics. To resolve the ambiguity, stand-alone array names retain their existing C semantics (as base pointers) and all "wild card" vector references must be explicitly coded.

Example 5 shows a vector multiply expression. Vector expressions have a similar syntax to their Fortran 8X counterparts. The standard arithmetic operators (+, -, *, /, ++, - -, etc.) are overloaded to handle vector operations. The C logical operators are also overloaded to support operations on vector operands. Scalars intermixed with vectors are expanded to the appropriate shape, as necessary. Note, however, the interesting operation of Example 6. As with Example 3, the right-hand side of the assignment is not conformable with the left-hand side. Additionally, the operands of the right-hand side are also not conformable with each other. Again, in keeping with VPC's complacent attitude, this statement is not rejected by the compiler. Instead, the default shape for the computation is the same as the shape of the left-hand side of the assignment statement. For the statement in Example 6, the first 100 elements of

[5] As is allowed by Fortran 8X.

the c array are multiplied by the first 100 elements of the b array and assigned to the a array. In those cases where the length of the target array is longer than one or more of the operands, unpredictable values will result. Although VPC could define a zero fill default (or some other default that is appropriate for the specific data type) for such cases, the resulting run-time code would be less efficient. Preference is given to performance rather than safety for the expected case (of correct programs). VPC generates a compile-time warning for expressions and assignments that are known to be non-conformable at compile time.

### 3.3. Vector Constants (Array Constructors)

VPC supports the specification of vector constants. The syntax for this is the same as the vector and structure initialization syntax for standard C programs. For example, the C language currently permits initialization of an array in a type declaration statement as follows:

```
int x[10] = {
      1, 1, 1, 2, 2, 2, 3, 3, 3, 4
      };
```

VPC extends that concept to allow vector constants to be specified within executable statements:

```
int x[5], y[5];
.
.
x[0:4] = y[0:4] + {1, 2, 3, 3, 3};
```

As expected, this sets x[0] = y[0] + 1, x[1] = y[1] + 2, etc.

A triplet notation is also supported and has the following syntax:

```
int       x[5];
x[] = {1:9:2};
```

which gives x[0] through x[4] the values of 1, 3, 5, 7, and 9, respectively. Triplets have the same format as vector subscripts. Again, the third field (*stride*) and its preceding colon are optional and, if missing, defaults to one.

# 4. NON–LOOP PARALLEL CONSTRUCTS

## 4.1. Overview

The initiation of parallel execution streams is one of the fundamental extensions offered by VPC. These streams may be started through slower, conventional system calls such as Unix fork(), or may be handled by more efficient means such as "microtasking" from Cray Research (Ref. 6) and Sequent (Ref. 14), or "threads" in the Mach operating system from Carnegie Mellon (Ref. 1). Although the manner in which the new streams are started does not affect the basic parallel constructs of VPC, it does affect the level of parallelism granularity that may be used before all benefit is lost due to overhead. The parallelism extensions of VPC have been designed to take advantage of an efficient, low–overhead tasking mechanism.

## 4.2. The COEXEC Statement

VPC provides explicit parallelism to the programmer through a single program construct, the COEXEC statement. The syntax is as follows:

coexec([*ezpr*]) *stmt*

The rules specifying the semantics of this construct are defined as follows. Each instance of a COEXEC statement within a program results in the initiation of an independent thread[6] of execution on another (possibly virtual) processor.[7]

*Stmt* is any C statement, including a block statement (a list of statements surrounded by braces). *Stmt* represents the code that is to be executed in parallel. This may consist of code at any level of granularity, from a single assignment statement to a block statement comprising several function calls. No identifiers that are referenced in this statement may be declared as private, or a compile–time type error is reported. All code specified in this statement is treated as a single execution thread and executed on a

[6] "Thread" and "stream" are used interchangeably throughout this paper.

[7] Virtual processor in this context is defined as follows. If sufficient resources exist at run time, an idle processor is assigned to satisfy the request. If not, the request is satisfied by assigning a busy processor and multiplexing the work load on that processor. All references to processors in this specification are intended to mean virtual processors.

single virtual processor.

*Expr* can be any valid C arithmetic expression that is evaluated and treated as a boolean guard. This expression is optional, and indicates the conditions under which the new stream will begin executing. If *expr* is omitted, the empty set of parentheses must still be placed before *stmt*. No identifier used in *expr* may be declared as **private**, or a compile-time error results. If the expression is not specified in the COEXEC structure, *stmt* immediately becomes enabled for execution. If the expression does exist, the designated statement becomes enabled for execution as soon as the value of the expression becomes non-zero (boolean TRUE in C).

### 4.3. Non-Synchronized Parallelism

The spawning of a parallel execution thread through the use of the COEXEC statement does no inherent synchronization. After the appropriate initialization of the new processor has been performed, the originating processor continues execution with the statement immediately following the COEXEC statement. If the COEXEC statement specifies a non-null precondition expression, the checking for this expression is done by the spawned thread, not the originating one. This allows the originating processor to proceed with minimal delays while starting parallel elements in a VPC program.

For example, in the sequence:

```
.
.
coexec(y == 3)  x = 2;
f();
.
.
```

a second virtual processor is allocated to this job. Control is then immediately passed back to the original processor, at which point it begins executing the function, f(). Meanwhile, the new processor begins exe-

cuting in parallel.[8] The second processor initiates by waiting for the condition "y == 3" to become true. Once the condition is determined to be true, the assignment statement "x = 2" is executed. At that point the newly spawned thread terminates and its processor is deallocated.[9]

A few examples follow:

```
/* main-line program */
  .
  .
coexec () p();
coexec () q();
r();
  .
  .
```

The initial program is assumed to be running on a single processor. When the program reaches the first COEXEC statement, the routine p() is started on another processor while the spawning processor continues to execute. The original processor then spawns routine q() in a third processor. The original processor now continues by executing routine r() while p() and q() are executing on other processors. Note that this example shows *no* synchronization between any of the co-executing routines.

Here is a slightly different example:

```
  .
  .
coexec () {
        p();
        q();
}
r();
  .
  .
```

Here the original processor spawns a second thread of execution at the point of the COEXEC statement.

[8] Subject to the availability of resources and system-dependent restrictions.

[9] Conceptually, processors are allocated and deallocated on demand at run-time, but implementation overhead will probably dictate a somewhat more efficient strategy. This, however, should not affect the ensuing discussions.

Now, however, the statement that is specified in the COEXEC statement is a compound statement (indicated by the set of braces). This causes the cooperation of only one other processor in the execution. The second processor executes p() and q() sequentially while the first processor executes routine r(). Again, no synchronization is present.

Since the COEXEC construct may be applied to any C statement, VPC programs may achieve parallelism with very fine granularity.[10]

```
coexec () x = 3;
coexec () y = 4;
coexec () z = x;
```

In this case, all three assignment statements can take place in separate processors. Note that VPC will not warn the user about the potential race involving the assignment of **x** to **z** and the assignment of **3** to **x**. However, VPC insists that **x**, **y**, and **z** are not declared as **private** variables so that their values can be communicated between processors.

The COEXEC statement may be nested an arbitrary number of levels. Consider the following example:

```
.
.
coexec () {
        p();
        coexec () q();
        r();
}
s();
.
.
```

Here a new processor is started and begins execution of the code in routine p(). In the meantime, the first processor begins the execution of the routine s(). After the second processor completes the execution of p(), it starts the co-execution of a third processor on function q(). Meanwhile, the second processor continues with the execution of routine r().

[10] As mentioned earlier, tasking granularity is limited by system architecture and operating system overhead parameters.

### 4.4. Synchronization

VPC allows the synchronization of parallel constructs through a set of three intrinsic functions. These functions provide indivisible memory operations that allow the programmer to control access to shared variables through built-in language constructs. The reason for using built-in functions is that language-level constructs can be implemented efficiently, allowing programmers to avoid machine-dependent sequences in assembly language.

The synchronization functions are:

tstlock(*sync_var*)
setlock(*sched, sync_var*)
clrlock(*sync_var*)

The tstlock(), setlock(), and clrlock() intrinsics provide the programmer with the traditional test-and-set style of functionality. The *sync_var* argument for these functions is expected to be a sync variable which is a data object that is associated with a unique, dedicated lock field. This lock field is indivisibly manipulated by the intrinsics while the data fields are left unchanged. *Sync_var* must be declared as a sync variable or a compile-time error is generated. Tstlock() functions exactly as test-and-set. It is a boolean function that indivisibly tests the state of the specified lock and rewrites it as "locked." The function then returns true if the lock was set originally, or false if it was not (and was therefore set by the caller).

Setlock() is similar to tstlock() except that the caller is blocked until the lock can be set. Therefore, setlock() always returns false (0) to the caller. Since some operating systems may have efficient facilities for blocking synchronizing processors, setlock() is preferred over a busy-wait using tstlock() whenever possible.

With setlock(), another parameter, *sched*, is specified in order to give the programmer some control over the way scheduling and control are handled if the calling processor becomes blocked. If *sched* is SCH_BUSYW, the calling processor performs a busy wait loop until the specified locking operation can be completed. If *sched* is SCH_SWITCH, a context switch will occur if the originating processor becomes

blocked and there are other tasks waiting on the ready queue. Finally, a value of SCH_SLEEP for *sched* indicates that neither a busy wait nor a task switch is to be performed during the blocking condition. The calling processor is blocked and its processor remains allocated and dormant until the lock can be set.

The **clrlock()** function simply clears the lock associated with the specified sync variable. The programmer should be aware that the locking mechanism is advisory only. Nothing in the language prohibits a task from accessing a locked variable if the lock is not checked. VPC only guarantees that locked variables are not modified by assignment statements or unary operators (e.g., "++") *when the compiler is aware that the target variables are sync variables.* Passing the address of sync variables to separately-compiled VPC routines may conceal the variable's access class, causing the VPC compiler to omit generating code for checking lock status. Additionally, locking is enforced for writes only – all variable fetches are executed without regard to lock status.

Using the synchronization intrinsic functions, a flexible structure for coordinating processors is possible. In general, new threads of execution that need synchronization must execute some sequence of syn-

```
main()
{
        sync int t1;        /* NOTE — all identifiers  */
        int x, y, z;        /* shared by default       */

        t1 = 0;
        coexec() {
                x = f();
                t1++;
          }
        coexec() {
                y = g();
                t1++;
          }
        while (t1 != 2)
                ;                   /* do nothing */
          z = x + y;
}
```

Program 1

```
main()
{
        sync int t1 = 0;

        coexec() {
                a();
                t1++;
          }
        b();
        while (t1 == 0)
                ;                       /* do nothing */
        coexec()
                c();
        d();
}
```

Program 2

chronization functions just prior to termination. Then, awaiting threads may inspect the sync variables in their guard expressions in order to control execution.

Consider the VPC code in Program 1. The first processor spawns two threads. The first one computes function f() and assigns the value to x, and the second one computes function g and assigns the value to y. While these two threads are executing, the original processor enters a busy wait, testing for the completion of the two parallel threads. The integer t1 is declared as a sync variable, and has been introduced for the purpose of coordinating the three threads. T1 in this example has been designed to represent the number of parallel threads that have completed execution. The original processor sets this counter to zero at the beginning of the program. For synchronization, both COEXEC statements have been designed to increment this counter at the completion of their code sequences. When both threads have completed, t1 will be 2. Finally, the busy wait loop on t1 insures that the original processor will not attempt to compute the sum before the addends are ready.

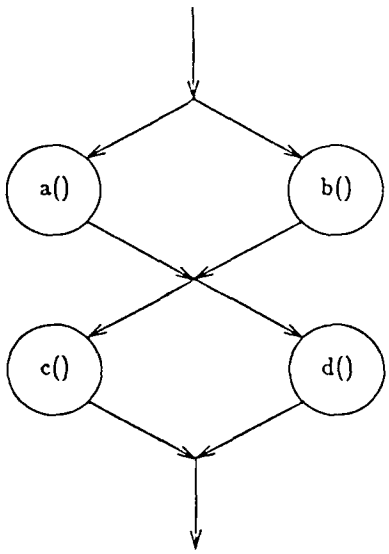



Figure 1 Computation Graph for Program 2

Program 2 shows a two-processor mapping of four function calls that corresponds to the computation graph shown in Figure 1. Functions a() and b() may be executed simultaneously and functions c() and d() may be executed simultaneously. However, neither c() nor d() may begin until both a() and b() have completed.

Since the guard expression in the COEXEC statement is a fully general C expression, the execution of any arbitrary computation graph can be realized. Consider the graph shown in Figure 2. This graph shows a more complex parallelism and synchronization structure that can be handled with the COEXEC statement in VPC. The graph shows that function a() must complete before functions b() and c() may begin. Additionally, function d() may begin after b() completes and function f() may begin after c() completes. Function e() may not begin until both b() and c() complete. Finally, function g() may begin after functions d(), e(), and f() have completed. One possible VPC encoding for this effect is shown in Program

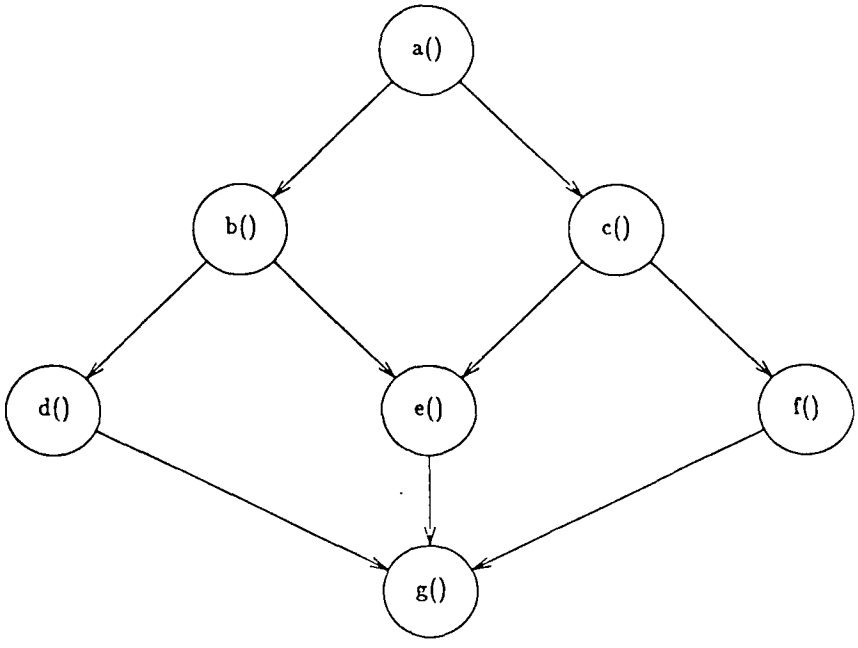



Figure 2 Computation graph with multiple synchronization points

3.[11]

### 4.5. Local Variables

VPC supports the declaration of processor local variables with the COEXEC statement. By including type declaration statements within a compound statement inside of the COEXEC construct, the programmer may initiate the allocation of a set of variables that will be local to that processor. These variables are visible only to the newly created thread and its offspring, and exist only while the spawned processor is executing. Program 4 initiates a new processor via the first COEXEC statement. Identifiers **x** and **y** are declared as shared (by default) in **main** and are therefore visible to the newly spawned processor. The first COEXEC, however, declares a processor local (but sharable) copy of **x** which conceals

[11] This example could be implemented using fewer synchronization variables. However, one variable is used for each function to simplify the illustration.

```
main()
{
        sync int bdone = 0,
                 cdone = 0,
                 ddone = 0,
                 edone = 0,
                 fdone = 0;

        a();
        coexec () {
                 b();
                 bdone++;
        }
        coexec () {
                 c();
                 cdone++;
        }
        coexec (bdone) {
                 d();
                 ddone++;
        }
        coexec (bdone && cdone) {
                 e();
                 edone++;
        }
        coexec (cdone) {
                 f();
                 fdone++;;
        }
        coexec (ddone && edone && fdone)
                 g();
}
```

Program 3

main's copy of **x** from this processor (although **y** is still visible). When the first COEXEC is executed, the new copy of **x** will be allocated.

The second processor now executes only one statement, namely, the second COEXEC. After the second COEXEC is executed, a third processor will be spawned to execute the three specified assignment statements. Meanwhile, the second processor terminates and deallocates its local copy of **x**. Note the potential danger using shared variables in this example. The second COEXEC "sees" the original declaration of **y** and the new declaration of **x** (since it is **shared** by default). If the second processor terminates

```
main()
{
        int x;
        int y;

        coexec() {
                int x;

                coexec() {
                        int a, b, c;

                        a = x + 1;
                        b = x + 2;
                        c = x + 3;
                }
        }
}
```

Program 4

before the third (which is likely in this example), the third processor will access the variable x that has been deallocated, causing indeterminate results. This problem can be cured by either using synchronization to prevent processor two from terminating prematurely, or declaring x tc be private in the first COEXEC statement. This latter solution will have the effect of making the original copy of x visible to the third processor (at the second COEXEC statement).

# 5. PARALLEL LOOPING CONSTRUCTS

## 5.1. The COLOOP statement

VPC provides a special construct for loops whose iterations are to be performed in parallel. This construct is the COLOOP statement and has the following syntax:

COLOOP (*#procs*; *id* = *array_exp*)
*stmt*

The number of processors to be applied to the execution of this loop is specified by *#procs*. This number includes the original processor that encountered the COLOOP statement. If *#procs* is 1, then only the original processor will work on the iterations of the loop body.[12] The COLOOP statement may be treated as a function call (e.g., x = coloop (....)) in which case it will return the maximum number of processors applied during execution of the loop body.[13] The programmer may request that all available processors be assigned by specifying zero in the *#procs* field.

Some machines have special purpose hardware that allows efficient execution of parallel loops automatically.[14] VPC compilers for such machines may opt to use this hardware in lieu of a software-based tasking subsystem where appropriate. In those cases where programmer control over the generation of code to use these hardware facilities is desired, compiler directives enclosed in comments may be used. However, VPC does not officially support such machine-specific extensions.

*Id* is an identifier that functions as the parallel loop index variable. *Id* takes on the values generated by *array_exp* and gives one to each iteration of the loop body. *Array_exp* is a series of elements comprising either an array section or an array constructor as described in section 3.

Iterations are guaranteed to be scheduled in the order that index values are specified. For example, in the following loop:

---

[12] However, execution may not be the same as in the serial case as control will still be handled by the underlying tasking mechanism.

[13] "High-water mark."

[14] For example, the Alliant FX/8 can apply up to eight processors to execute the iterations of a parallel loop in a self-scheduled manner (Ref. 3).

```
coloop(1; i = {1, 2, 2, 4}) {
        .
        .
        (loop body using i)
        .
}
```

only one processor is requested for execution. This processor will first execute the loop body with the index variable i being equal to one, then two, two, and four, in order. Multiple processors working on the loop will not change the order in which the iterations are scheduled. Differences in execution time for individual iterations, however, could affect the specific iterations that a particular participating processor receives for execution. The scheduling order of the iterations is explicitly defined by VPC in order to provide a deterministic execution environment that will aid the user to do proper synchronization between loop iterations.

*Stmt* may be either a single C source statement or a compound statement and represents the entire loop body for the COLOOP construct. Parallelism for this loop is achieved by automatically scheduling the loop iterations over the designated number of processors (or whatever subset was available from the run-time support environment). An implicit barrier exists at the end of the loop body. The statement following the COLOOP statement will not begin execution until the loop has terminated execution. This synchronization is automatic and need not be managed by the programmer.

## 5.2. Exiting Parallel Loops – COBREAK

Early termination of concurrent loops is accomplished with the COBREAK statement. A COBREAK executed during any iteration of a COLOOP construct causes the VPC run-time environment to deny all future scheduling of new iterations. Iterations that have already been scheduled are allowed to continue to completion. This definition, combined with the guaranteed ordering of iteration scheduling defined by the COLOOP construct, assures that a continuous sequence of iterations will be executed. While the index of the highest-numbered iteration that executes is nondeterministic, the programmer can be sure that all iterations of a lesser number have completed.

For example:

```
coloop (0; i = 1:100) {
        ...
        if (f(i)) cobreak;
        ...
}
```

If function f(i) becomes true on iteration 15, the programmer can be sure that every iteration from one to n has completed, where $15 \leq n \leq 100$. However, the value of n may differ in successive executions of the program.

### 5.3. Local Variables

VPC supports processor local variables in parallel loops in the same way as the COEXEC statement. Identifiers that are declared within the braces of a compound loop body will be allocated on a per–iteration basis. Consider the following example:

```
coloop (6; i = {1:100:2}) {

        int x = 3;
        int y[100];
        ...
}
```

This loop executes on six virtual processors with the index variable i taking on the values of 1, 3, 5, ... Each time an iteration is given to one of the 6 processors, a copy of the scalar x and the 100 element array y are allocated.[15] Additionally, x is initialized to 3 for every iteration. Note that default access classes still apply; therefore, x and y[] are **shared** variables and may be used in COEXEC statements within the body of the COLOOP statement. Their scope, however, is still limited to the enclosing braces of the compound statement as is dictated by conventional C semantics.

[15] This is the conceptual model. Practical implementations will probably optimize this operation by doing allocations only once and initializations at the scheduling of every iteration.

## 6. ACKNOWLEDGEMENTS

Several people provided helpful input into the design of VPC. Many thanks to Richard Barton, Perry Emrath, Tim McDaniel, Robert McGrath, David Padua, and David Sehr for taking time to discuss the various alternatives of the concurrency constructs. Special thanks to Dennis Gannon and Duncan Lawrie for taking time to point out deficiencies and improvements, and participating in all aspects of the design of VPC.

# REFERENCES


[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young. "Mach: A New Kernel Foundation for Unix Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition* (June, 1986).

[2] Alliant Computer Systems Corporation. "Concentrix C Handbook", No. 302-00004-B, Acton, Massachusetts, August, 1986.

[3] Alliant Computer Systems Corporation. "FX/Series Architecture Manual", No. 300-00001-B, Acton, Massachusetts, January, 1986.

[4] U. Banerjee. "Speedup of Ordinary Programs", Rpt. No. UIUCDCS-R-79-989, Ph.D. Thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, October, 1979.

[5] W. Chang and A. Norton. "VM/EPEX C Preprocessor User's Manual", IBM T. J. Watson Research Center, Yorktown Heights, New York.

[6] Cray Research Incorporated. "Multitasking Programmer's Reference Manual", Publication No. SN-0222, Mendota Heights, Minnesota, October, 1986.

[7] W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge, Massachusetts, 1986.

[8] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[9] B. Kernighan and R. Pike. *The Unix Programming Environment.* Prentice-Hall, 1984.

[10] D. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, 1982.

[11] K. Li. "Vector C - A Programming Language for Vector Processing", Ph.D. Thesis, Purdue University, West Lafayette, Indiana, December, 1984.

[12] K. Li and H. Schwetman. "Vector C: A Vector Processing Language," *Journal of Parallel and Distributed Computing* (May, 1985), Vol. 2, No. 2, pp. 132-69.

[13] M. Metcalf. "Fortran 8X - The Emerging Standard," *ACM Fortran Forum* (April, 1987), Vol. 6, No. 1.

[14] A. Osterhaug. "Guide to Parallel Programming on Sequent Computer Systems", Sequent Computer Systems, Incorporated, Beaverton, Oregon, 1986.

[15] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton and a. J. Weiss. "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing* (August, 1985), pp. 764-771.

[16] Sequent Computer Systems Incorporated. "Balance Tecnical Summary", No. MAN-0110-00, Beaverton, Oregon, November 19, 1986.

[17] Thinking Machines Corporation. "Introduction to Data Level Parallelism", Rpt. No. 86.14, April, 1986.

[18] University of California. *UNIX User's Manual, Reference Guide--4.2 Berkeley Software Distribution.* Computer Science Division, University of California, Berkeley, California, 1984.

[19] M. Wolfe. "Optimizing Compilers for Supercomputers", Rpt. No. UIUCDCS-R-82-1105,

Ph.D. Thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, October, 1982.

END

DATE

FILMED

5-88

DTIC